AI ASSISTED CODING LAB TEST-1

NAME:N.VISHWAS

2403A510E4

BATCH-05

**Q1. Zero-shot Classification [5M]**

**• Scenario: Classify tweets into “Positive,” “Negative,” or “Neutral.”**

**• Task 1: Write a zero-shot prompt to classify sentiment without any examples.**

PROMPT : Classify the sentiment of the following tweet as either “Positive,” “Negative,” or “Neutral.” Provide only the sentiment label.

Tweet: “Just finished my final exam. Feeling relieved but exhausted.”

CODE : tweet = "Just finished my final exam. Feeling relieved but exhausted."

positive\_words = ["relieved", "finished", "final", "accomplished", "happy", "good"]

negative\_words = ["exhausted", "tired", "bad", "sad", "worried", "stressed"]

def classify\_sentiment(text):

text\_lower = text.lower()

pos\_score = sum(word in text\_lower for word in positive\_words)

neg\_score = sum(word in text\_lower for word in negative\_words)

if pos\_score > neg\_score:

return "Positive"

elif neg\_score > pos\_score:

return "Negative"

else:

return "Neutral"

sentiment = classify\_sentiment(tweet)

print(f"The sentiment of the tweet is: {sentiment}")

OBSERVATION:

 The code counts positive and negative words in the tweet.

 Positive words: 3 → finished, final, relieved

 Negative words: 1 → exhausted

 **Result:** More positive words → Sentiment is **Positive**.

**Task 2: Create a scenario where an AI assistant needs to help a student solve math problems.**

**Write two prompts: one without context and one with detailed context (e.g., grade level, topic,**

**difficulty).**

**PROMPT(WITHOUT CONTEXT):**

generate a python code for finding amstrong number.

**CODE:**

def is\_armstrong(number):

digits = [int(d) for d in str(number)]

power = len(digits)

total = sum(d \*\* power for d in digits)

return total == number

# Example usage:

num = int(input("Enter a number: "))

if is\_armstrong(num):

print(f"{num} is an Armstrong number.")

else:

print(f"{num} is not an Armstrong number.")

**OUTPUT:**

Enter a number:153

153 is a amstrong number

**OBSERVATION:**

The function is\_armstrong(number) checks if a given number is an Armstrong number (also called a narcissistic number).

It works by splitting the number into its digits, raising each digit to the power equal to the total number of digits, and summing them.

If this sum equals the original number, then it is an Armstrong number.

**PROMPT(WITH CONTEXT):**

"Write a Python program to check whether a given number is an Armstrong number. The program should take an integer input from the user, split it into digits, raise each digit to the power of the number of digits, and calculate the sum. If the sum equals the original number, print that it is an Armstrong number; otherwise, print that it is not."

**CODE:**

   def is\_armstrong(number):

digits = [int(d) for d in str(number)]

power = len(digits)

total = sum(d \*\* power for d in digits)

return total == number

num = int(input("Enter a number: "))

if is\_armstrong(num):

print(f"{num} is an Armstrong number.")

else:

print(f"{num} is not an Armstrong number.")

**OUTPUT:**

Enter a number: 153

153 is an Armstrong number.

**OBSERATION:**

The function is\_armstrong(number) converts the number into digits and raises each digit to the power of the total number of digits.

It then sums these powered values and compares the result with the original number.

If they match, the number is an Armstrong number, else it is not.

--------------------------------------------------------------------------------------------------------------

**Q2. One-shot vs Few-shot [5M]**

**• Task 1:**

**o A one-shot prompt**

**PROMPT:**

generate a python code to find prime numbers in a specified range

example

input::10

output:1 3 5 7

**CODE GENERATED:**

def find\_primes(n):

primes = []

for num in range(1, n + 1):

if num < 2:

if num == 1:

primes.append(1)

continue

for i in range(2, int(num \*\* 0.5) + 1):

if num % i == 0:

break

else:

primes.append(num)

return primes

n = int(input("input::"))

result = find\_primes(n)

print("output:", ' '.join(str(x) for x in result))

**OUTPUT:**

Input : 10

Output:1 2 3 5 7

**OBSERVATION:**

This code defines a function find\_primes(n) that generates all numbers up to n which it treats as prime. It checks divisibility of each number up to its square root and appends primes to a list. However, it also includes 1 as a prime, which is mathematically incorrect. The logic for checking primes is correct for numbers ≥ 2, but the print statement at the end is incomplete and will cause an error unless fixed.

**O A few-shot prompt**

**PROMPT:**

generate a python code to find prime numbers in a specified range

example

input::10

output:1 3 5 7

input:20

output:1 2 3 5 7 11 13 17 19

input:30

output:1 2 3 5 7 11 13 17 19 23 29

**GENERATED CODE:**

def find\_primes(n):

    primes = []

    for num in range(1, n + 1):

        if num == 1:

            primes.append(1)

        elif num == 2 or num == 3:

            primes.append(num)

        else:

            for i in range(2, int(num \*\* 0.5) + 1):

                if num % i == 0:

                    break

            else:

                primes.append(num)

    return primes

n = int(input("input::"))

result = find\_primes(n)

print("output:", ' '.join(str(x) for x in result))

**OUTPUT:**

Input: 30

Output:1 3 5 7 11 13 17 19 23 29

**OBSERVATION:**

This program finds and prints all prime numbers up to a given input `n`. It checks each number for divisibility up to its square root, and if no divisor is found, the number is considered prime and added to the list. However, the code also includes `1` as a prime, which is mathematically incorrect since prime numbers are defined as being

greater than 1. The logic otherwise works correctly for numbers ≥ 2, and the output displays the primes (along with `1`) separated by spaces.

**Task 2: Compare outputs on the same set of tweets and explain the difference.**

**COMPARISION BETWEEN OUTPUTS:**

Outputs are the same for any input n. Both codes give primes + 1.

Difference in approach:

-Code 1 uses a general loop for all numbers ≥ 2.

-Code 2 uses a few explicit cases (1, 2, 3) and then a loop for the rest.

Efficiency: Code 2 avoids unnecessary loop checks for 2 and 3, but this difference is tiny for small n.

Both codes still have the same flaw: including 1 as prime, which is mathematically incorrect.